



September 2026

Prepared for  
Tenka

Audited by  
pandadevi  
devtooligan

# Tenka Smart Contract Security Audit

Smart Contract Security Assessment

## Contents

|          |                                                                                                |          |
|----------|------------------------------------------------------------------------------------------------|----------|
| <b>1</b> | <b>Review Summary</b>                                                                          | <b>2</b> |
| 1.1      | Protocol Overview . . . . .                                                                    | 2        |
| 1.2      | Audit Scope . . . . .                                                                          | 2        |
| 1.3      | Risk Assessment Framework . . . . .                                                            | 2        |
| 1.3.1    | Severity Classification . . . . .                                                              | 3        |
| 1.4      | Key Findings . . . . .                                                                         | 3        |
| 1.5      | Overall Assessment . . . . .                                                                   | 3        |
| <b>2</b> | <b>Audit Overview</b>                                                                          | <b>4</b> |
| 2.1      | Project Information . . . . .                                                                  | 4        |
| 2.2      | Audit Timeline . . . . .                                                                       | 4        |
| 2.3      | Audit Resources . . . . .                                                                      | 4        |
| 2.4      | Critical Findings . . . . .                                                                    | 6        |
| 2.5      | High Findings . . . . .                                                                        | 6        |
| 2.6      | Medium Findings . . . . .                                                                      | 6        |
| 2.6.1    | Outstanding reservations can strand deposits after amortisation begins . .                     | 6        |
| 2.6.2    | Partial-fill rounding can consume bids without transferring shares . . . .                     | 7        |
| 2.6.3    | Partial principal return can be sold again before the Oracle resets dirty NAV                  | 8        |
| 2.6.4    | Unbound virtual mint and Oracle updates can misprice Bookbuild allocations                     | 9        |
| 2.6.5    | Routine operator can create trusted rogue vaults and drain funds . . . . .                     | 11       |
| 2.6.6    | Revoked KYC holders can liquidate DealVault positions through the<br>MatchingEngine . . . . .  | 13       |
| 2.7      | Low Findings . . . . .                                                                         | 14       |
| 2.7.1    | ERC-1155 callback gas grieving can block Bookbuild processing . . . . .                        | 15       |
| 2.7.2    | Zero-NAV turbo claims revert on yield-only distributions . . . . .                             | 15       |
| 2.7.3    | Partial Agora swaps leave vault funds exposed through the public swap<br>function . . . . .    | 16       |
| 2.7.4    | Reservation replacement can authorize two allocations . . . . .                                | 18       |
| 2.7.5    | MasterVault shares can be transferred to and liquidated by non-KYC<br>addresses . . . . .      | 19       |
| 2.7.6    | Sparse tick scanning can require 101 refund transactions . . . . .                             | 20       |
| 2.7.7    | Compromised operator can drain FundingVault assets through max-spread<br>round trips . . . . . | 22       |
| 2.7.8    | Negative MMF returns can lock bookbuild bids and subscribed rounds . .                         | 23       |
| 2.8      | Gas Savings Findings . . . . .                                                                 | 24       |
| 2.9      | Informational Findings . . . . .                                                               | 25       |
| 2.9.1    | Harden deployment and vault-creation validation . . . . .                                      | 25       |
| 2.9.2    | Remove the unused NAV_PENDING state . . . . .                                                  | 26       |
| 2.9.3    | Outstanding reservations can capture previously accrued yield . . . . .                        | 26       |
| 2.9.4    | EIP-7702 delegation lets a malicious operator exceed the Bookbuild target                      | 27       |
| 2.9.5    | DealVault Does Not Enforce Matching Token Decimals . . . . .                                   | 28       |

## 1 Review Summary

### 1.1 Protocol Overview

Tenka is a tokenized private-credit platform that lets investors access off-chain deals through on-chain vaults. A MasterVault pools stablecoin liquidity and allocates it to per-deal FundingVaults, which in turn take positions in tranche-based DealVaults representing individual raises. Pricing is driven by an Oracle publishing periodic NAV checkpoints, primary allocation runs through a Bookbuild auction, and all primary and secondary trading is KYC-gated and executed through a central MatchingEngine. An MMF integration (via an Agora/Uniswap V3 helper) puts idle FundingVault liquidity to work between raises.

### 1.2 Audit Scope

This audit covers 12 smart contracts and the mainnet deployment script across the DealVault, MasterVault/FundingVault, MatchingEngine, Oracle, and Bookbuild modules, reviewed over 9 days from September 1 to September 11, 2026.

```
packages/smart-contracts/src
├── BaseVault.sol
├── Oracle.sol
├── MatchingEngine.sol
├── DealVault
│   ├── DealVault.sol
│   ├── DealVaultFactory.sol
│   ├── DealVaultBookbuild.sol
│   └── Queue.sol
├── MasterVault
│   ├── MasterVault.sol
│   ├── TenkaFundingFactory.sol
│   ├── FundingVault.sol
│   ├── MMFManager.sol
│   └── AgoraHelper.sol
```

### 1.3 Risk Assessment Framework

### 1.3.1 Severity Classification

| Severity             | Description                                                                                                                                                                                                                                                                                   | Potential Impact                                    |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------|
| <b>Critical</b>      | Immediate threat to user funds or protocol integrity                                                                                                                                                                                                                                          | Direct loss of funds, protocol compromise           |
| <b>High</b>          | Significant security risk requiring urgent attention                                                                                                                                                                                                                                          | Potential fund loss, major functionality disruption |
| <b>Medium</b>        | Important issue that should be addressed                                                                                                                                                                                                                                                      | Limited fund risk, functionality concerns           |
| <b>Low</b>           | Minor issue with minimal impact                                                                                                                                                                                                                                                               | Best practice violations, minor inefficiencies      |
| <b>Undetermined</b>  | Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation. | Varies based on actual severity                     |
| <b>Gas</b>           | Findings that can improve the gas efficiency of the contracts.                                                                                                                                                                                                                                | Increased transaction costs                         |
| <b>Informational</b> | Code quality and best practice recommendations                                                                                                                                                                                                                                                | Reduced maintainability and readability             |

Table 1: severity classification

## 1.4 Key Findings

### Breakdown of Finding Impacts

| Impact Level                                      | Count |
|---------------------------------------------------|-------|
| <span style="color: red;">■</span> Critical       | 0     |
| <span style="color: orange;">■</span> High        | 0     |
| <span style="color: yellow;">■</span> Medium      | 6     |
| <span style="color: green;">■</span> Low          | 8     |
| <span style="color: gray;">■</span> Informational | 5     |

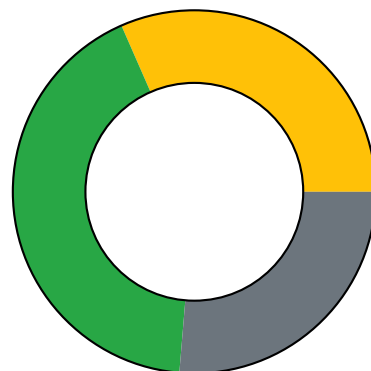


Figure 1: Distribution of security findings by impact level

## 1.5 Overall Assessment

Tenka is a well-built, thoughtfully engineered codebase backed by a responsive and knowledgeable team. Findings from this review concentrated in the interactions between independently reasonable subsystems: Oracle timing versus distributions, rounding at

settlement boundaries, and the trust boundary between factory-created vaults and the contracts that rely on them, rather than in any single component. The team engaged closely throughout, confirmed intended behavior quickly, and shipped fixes with regression tests for nearly every finding before this report was finalized. The protocol is intentionally centralized around a small set of operator and strategic roles that control matching, NAV publication, and vault creation. This is a deliberate design choice for a compliance-gated private-credit product, and it meaningfully reduces the on-chain attack surface compared to a permissionless design, but it also means the security of user funds ultimately depends on the operational security and honesty of those privileged roles, as reflected in the operator-trust findings in this report.

## 2 Audit Overview

### 2.1 Project Information

**Protocol Name:** Tenka

**Repository:** <https://github.com/tenka-fi/10k>

**Commit Hash:** d5b2750f10790225f78b285b6642593015b3cdf2

**Commit URL:**

<https://github.com/tenka-fi/10k/commit/d5b2750f10790225f78b285b6642593015b3cdf2>

### 2.2 Audit Timeline

The audit was conducted from September 1 to 11, 2026.

### 2.3 Audit Resources

- Code repository

| Category                 | Mark    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Access Control           | Average | Role checks within individual contracts were sound, but the boundary between privileged roles and the contracts that trust them had gaps: a routine operator role could create attacker-owned vaults that downstream contracts trusted automatically, and KYC status was not always re-checked at the point where value actually moved (revoked sellers, non-KYC transfer recipients). All of these were fixed by separating factory authority onto timelocks/multisigs and adding payer-side KYC checks.                                                                                                        |
| Mathematics              | Good    | Core accounting was correct, but several settlement paths did not handle boundary conditions: zero-output rounding in partial order fills, negative MMF returns underflowing cancellation math, and a division-by-zero on zero-NAV turbo claims. These were rounding and edge-case gaps rather than fundamental pricing errors, and most were remediated with explicit checks.                                                                                                                                                                                                                                   |
| Complexity               | Average | The protocol composes several independently reasonable subsystems (Oracle NAV publication, Bookbuild auctions, tranche accounting, an MMF yield strategy) whose cross-contract timing assumptions were not always enforced on-chain. Most findings in this report are ordering/sequencing issues at these seams rather than bugs within a single contract, which is typical of a system this size and was mitigated through a mix of code changes and documented operational procedure.                                                                                                                          |
| Libraries                | Good    | The codebase builds on well-established primitives (OpenZeppelin access control/ERC standards, Uniswap V3 for the MMF swap path) and uses them correctly in the vast majority of cases. The one notable gap, an unrestricted public swap() combined with unvalidated partial-fill deltas in the Uniswap V3 helper, was fixed by sweeping residual balances after each swap.                                                                                                                                                                                                                                      |
| Decentralization         | Average | Tenka is intentionally centralized: NAV publication, order matching, and vault creation are controlled by a small set of operator and strategic roles rather than being permissionless. This is a reasonable trade-off for a KYC-gated private-credit product and reduces on-chain attack surface, but it concentrates trust in those roles, as illustrated by the rogue-vault-creation and compromised-operator round-trip findings. Post-audit changes moved factory and vault-creation authority behind timelocks/multisigs, which meaningfully tightens this without removing the underlying centralization. |
| Documentation            | Good    | Inline documentation of invariants (e.g. matching token decimals across a DealVault) was present and accurate where it existed, and the team was consistently able to confirm intended behavior quickly and precisely during the review, indicating strong internal understanding of the system's design.                                                                                                                                                                                                                                                                                                        |
| Testing and verification | Good    | The existing test suite was a solid baseline, and the team was receptive to auditor-supplied proof-of-concept tests, several of which were incorporated directly.                                                                                                                                                                                                                                                                                                                                                                                                                                                |

## 2.4 Critical Findings

None.

## 2.5 High Findings

None.

## 2.6 Medium Findings

### 2.6.1 Outstanding reservations can strand deposits after amortisation begins

An outstanding reservation can be exercised at any point while a DealVault is `ACTIVE`. A reservation holder can exercise it immediately before the vault moves to `AMORTISING`, leaving the new deposit permanently stranded in the vault.

#### Technical Details

`takeReservation()` has no deadline and remains available while the DealVault is `ACTIVE`. Exercising a reservation transfers the user's funds into the DealVault and adds them to `trancheAmountRaised`.

Only `start()` and `reraise()` sweep `trancheAmountRaised` to the deal receiver. Once `setToAmortising()` changes the vault state, `reraise()` is no longer available. `finalize()` distributes separately returned capital but does not sweep or refund the unsent raised amount. The reservation deposit therefore remains in the DealVault without a path for the user or existing holders to claim it.

#### Impact

The full amount deposited through the late reservation can become permanently inaccessible. The path is confined to one DealVault and requires a pre-existing reservation to be exercised immediately before amortisation, which makes occurrence rare. The permanent loss of a potentially material deposit supports Medium severity.

#### Recommendation

Add a deadline to each reservation and reject `takeReservation()` after it expires. As a backstop, have `setToAmortising()` iterate through the tranches and revert if any tranche has a raised amount greater than zero.

#### Developer Response

Mitigated in [PR #320](#), commit [48809957](#). Reservations expire seven days after being set and must be renewed to remain usable. `setToAmortising()` also rejects transitions while any supported token/tranche has unsent raised capital. Regression tests cover reservation expiry and amortisation after a reservation deposit.

### 2.6.2 Partial-fill rounding can consume bids without transferring shares

When a bid cannot purchase the entire ask, `MatchingEngine._handleOrder()` rounds the corresponding instrument amount down. If the result is zero, the engine nevertheless consumes the bid's entire remaining asset amount and transfers it to the seller. The ask is reduced by zero and can therefore be reused against additional bids.

#### Technical Details

The partial-fill branch calculates the shares transferred to the buyer as follows:

```
1 if (assetAmount >= _bidExecution.remainingAmount) {
2   _bidExecution.status = OrderStatus.FILLED;
3   instrumentAmount = _bidExecution.remainingAmount * instrumentAmount / assetAmount;
4   assetAmount = _bidExecution.remainingAmount;
5   _bidExecution.remainingAmount = 0;
6   _askExecution.remainingAmount -= instrumentAmount;
7 }
```

The division can produce `instrumentAmount == 0`. Despite this, the bid is marked filled and `_processAccounting()` transfers `assetAmount` from the buyer to the seller. Since zero is subtracted from the ask, the seller's same share balance and order remain available for further matches.

Minimum order sizes do not prevent the attack. An attacker can first post an above-minimum MasterVault ask and use an ordinary partial match to leave one raw share unit outstanding. After victims post and fully fund above-minimum bids, the attacker can directly donate assets to the MasterVault so that this indivisible share unit is worth slightly more than each bid. Each subsequent match then rounds the purchased shares to zero.

The proof of concept in `test/audit/MatchingEngineZeroShareDrain.t.sol` demonstrates two victims each posting and funding a 1,000,000 USDC bid before the donation. The attacker donates 2,000,000 USDC, causes both bids to be filled for zero shares using the unchanged one-unit ask, and finally redeems that share. The attacker finishes approximately 1,000,000 USDC ahead. The attack requires no malicious Oracle, nonstandard token, or unusual operator-selected partial-fill amount.

#### Recommendation

Reject any match for which the calculated `instrumentAmount` is zero. More generally, after rounding the instrument amount, recompute the asset amount owed for the shares actually transferred instead of consuming the bid's full remaining balance. Leave unspent bid assets in the order or return them to the buyer.

Add regression coverage for zero-share partial fills, high share prices, and below-minimum remainders created from otherwise valid orders.

#### Developer Response

The zero-output loss path is addressed in <https://github.com/tenka-fi/10k/pull/320>, commit [689cf4aa](#). Matching now rejects zero instrument or asset amounts before consuming orders or

transferring proceeds. Regression tests cover dust remainders and donation-induced zero-share fills.

The same commit adds operator-provided P2P fill limits and independent order completion. The broader recommendation to recompute payment after nonzero share rounding is not implemented.

### 2.6.3 Partial principal return can be sold again before the Oracle resets dirty NAV

`DealVault.updateYield()` credits a partial principal return to the existing holders without updating the Oracle or blocking secondary settlement. Until the next Oracle update, `convertToAssets()` and the `MatchingEngine` continue pricing the shares using the previous dirty NAV. A seller can keep the distributed principal and sell the remaining shares as though that principal were still part of the position.

#### Technical details

The project team confirmed that each tranche's Oracle feed contains dirty NAV and that the accumulated portion resets after a distribution. They also confirmed that the intended sequence is to update the Oracle before distributing principal. The code-freeze contracts do not enforce that sequence. The Oracle update, principal distribution, and secondary match are separate transactions, with no pending-distribution flag or checkpoint preventing settlement between them.

Example exploit:

1. Creates a tranche with one million shares and a \$1 million dirty NAV.
2. Distributes \$400,000 of principal to the existing holder.
3. Sells 240,000 shares before publishing the lower ex-distribution NAV.
4. The `MatchingEngine` charges the buyer \$240,000 because the old \$1 million NAV remains active.
5. Those shares represent only \$144,000 of the remaining \$600,000 principal, leaving the buyer with a \$96,000 loss.
6. The seller retains the previously credited \$400,000 principal return.

The control publishes the \$600,000 ex-distribution NAV before matching. The same 240,000 shares then cost \$144,000 and the loss disappears.

#### Affected Code

- `packages/smart-contracts/src/DealVault/DealVault.sol`: `updateYield()`, `convertToAssets()`, `transferDeal()`
- `packages/smart-contracts/src/MatchingEngine.sol`: secondary-order settlement
- `packages/smart-contracts/src/Oracle.sol`: tranche dirty-NAV publication

#### Impact

A buyer can pay for principal that has already been distributed to the seller. The demonstrated scenario causes a \$96,000 loss on a \$240,000 purchase.

## Recommendation

Block secondary settlement and new allocations after a principal or yield distribution until a newer Oracle checkpoint clears the pending valuation change. The contracts should enforce this relationship rather than relying only on transaction ordering by the operator.

## Developer Response

Acknowledged and accepted. NAV updates and yield/principal distributions will be coordinated within our weekly operational cycle. We will suspend matching for affected DealVaults during any gap between the NAV adjustment and the corresponding distribution, including payment delays. Trading will resume only once both are consistent. This restriction will be enforced operationally by our matching operators; no on-chain change is planned.

### 2.6.4 Unbound virtual mint and Oracle updates can misprice Bookbuild allocations

`virtualStakeFor()` mints shares without transferring assets or binding the mint to the Oracle update that represents the off-chain investment. If virtual shares are minted before that update, a Bookbuild allocation processed in the gap receives too many shares and captures value from the existing and virtual investors.

## Technical Details

The updated `virtualStakeFor()` correctly accepts the off-chain-computed share amount directly, but it only increases supply and mints the shares:

```
1 function virtualStakeFor(address _user, uint256 _shares, uint96 _trancheMask)
2     external
3     onlyStrategic
4     notEmergency
5     kycOrFundingVault(_user)
6 {
7     // ...
8     TrancheData storage td = trancheData[_trancheMask];
9     td.totalSupply += _shares;
10    td.updateAccounting(_user, balanceOf(_user, _trancheMask));
11    _mint(_user, _trancheMask, _shares, "");
12 }
```

The corresponding Oracle update is a separate transaction. Bookbuild allocations call `convertToShares()`, which uses the latest Oracle NAV and the current total supply:

```
1 function convertToShares(uint256 _assets, uint96 _trancheMask) public view returns (
2     uint256) {
3     uint256 cleanNav = _getNonStaleCleanNav(_trancheMask) + _getRaisedAmount(
4         _trancheMask) + 1;
5     uint256 totalSupply = trancheData[_trancheMask].totalSupply + 1;
6     return Math.mulDiv(_assets, totalSupply, cleanNav, Math.Rounding.Floor);
7 }
```

If the virtual mint happens first, total supply includes the new virtual shares while NAV does not yet include the off-chain contribution. A Bookbuild bidder entering during that interval is therefore issued shares at an artificially low price.

Starting with 100 shares representing \$100, the flow then mints 100 virtual shares for a \$100 off-chain investment. A \$100 Bookbuild allocation processed before the Oracle update receives approximately 200 shares. After the Oracle is corrected and \$300 is distributed, the Bookbuild bidder receives approximately \$150 while the original and virtual investors receive approximately \$75 each. Publishing the correct Oracle value before Bookbuild processing results in approximately \$100 for each investor.

The same ordering issue was reproduced after a normal raise and transition to **ACTIVE**. Because the Oracle permits only one tranche update per week, a corrective update can also be unavailable until the following week.

### Affected Code

- `packages/smart-contracts/src/DealVault/DealVault.sol`: `virtualStakeFor()`, `convertToShares()`
- `packages/smart-contracts/src/DealVault/DealVaultBookbuild.sol`: `processBids()`
- `packages/smart-contracts/src/Oracle.sol`: tranche NAV checkpoints

### Impact

Medium. The demonstrated ordering transfers 25% of the original and virtual investors' value to the intervening Bookbuild bidder. The loss is permanent and scales with the affected investments. The likelihood is reduced because the strategic service must call `virtualStakeFor()` before the required Oracle update and Bookbuild processing must occur before correction.

### Recommendation

Bind every virtual mint to the specific Oracle checkpoint that includes its off-chain contribution. Reject a virtual mint if that checkpoint is missing, stale, or already consumed for incompatible issuance. Prevent Bookbuild and other share-producing allocations from executing while a virtual issuance lacks a valid valuation checkpoint.

### Developer Response

Acknowledged. To mitigate that, offchain rules will be set up:

- No new allocations into the affected DealVault tranche from the valuation snapshot until the off-chain mint completes. This includes bookbuild processing and FundingVault staking.
- Once the oracle includes the off-chain investment, mint all corresponding shares before any secondary order matching resumes for that tranche.

### 2.6.5 Routine operator can create trusted rogue vaults and drain funds

The mainnet deployment gives the routine `operator` enough authority to create attacker-owned vaults that the protocol immediately trusts. A malicious authorized operator can use either factory boundary to drain protocol assets:

1. create a rogue `FundingVault` and drain `MasterVault` assets through a malicious MMF helper; or
2. create a rogue `DealVault` whose owner confiscates and repeatedly resells one legitimately backed `FundingVault` position.

Both paths arise because a factory operator can choose the new vault's owner and factory registration is sufficient for downstream contracts to trust that vault.

#### Technical details

The deployment grants the same routine address roles across both paths:

```
1 tenkaFundingFactory.grantRole(operator, OPERATOR_ROLE, true);
2 masterVault.grantRole(operator, OPERATOR_ROLE, true);

4 dealVaultFactory.grantRole(operator, OPERATOR_ROLE, true);
5 matchingEngine.grantRole(operator, OPERATOR_ROLE, true);
6 fundingVault.grantRole(operator, OPERATOR_ROLE, true);
```

Transferring factory and vault ownership to timelocks does not revoke these operator grants.

#### Path 1: rogue `FundingVault` drains the `MasterVault`

`TenkaFundingFactory.createVault()` lets a factory operator choose the new `FundingVault`'s owner. It also registers the child, so `MasterVault.allocateFunds()` accepts it as a valid destination.

The operator can:

1. create a registered `FundingVault` with an attacker-controlled owner;
2. use that ownership to grant child-local roles and configure a malicious MMF helper;
3. allocate `MasterVault` assets to the child; and
4. process an MMF request that executes the helper with `delegatecall` and transfers the allocated assets out.

The original PoC transferred 500,000 USDC from a `MasterVault` holding 1,000,000 USDC. Revoking only the routine operator's `TenkaFundingFactory` role stopped this path.

#### Path 2: rogue `DealVault` owner recycles one backed position

`DealVaultFactory.createVault()` also lets its operator choose any owner and immediately registers the child. After the legitimate `FundingVault` allocates 100,000 USDC to that child, it receives 100,000 legitimately backed shares.

However, `DealVault.transferDeal()` lets the child owner move shares out of any holder without that holder's approval. The attack is therefore:

1. create a registered DealVault owned by a colluding KYC account;
2. allocate 100,000 USDC from the legitimate FundingVault to establish a fully backed position;
3. have the child owner confiscate those shares from the FundingVault;
4. sell the same shares back to the FundingVault at truthful par; and
5. repeat the confiscate-and-resell cycle nine times during one ordinary settlement window.

The PoC gives the malicious operator no Oracle role. A separate honest Oracle publishes the truthful zero off-chain clean NAV for the fresh OPEN deal; the conversion includes the 100,000 USDC of on-chain raised capital. The FundingVault uses zero price spread, needs no Backstop LP status, and uses no approved receiver path. The colluder supplies no USDC.

After nine par matches, the FundingVault has spent its entire 900,000-USDC idle balance. The colluder receives 899,100 USDC after fees, while the FundingVault retains only the original 100,000-USDC-backed position. The attack needs no virtual mint or child strategic role. Removing the child owner's third-party transfer authority makes the unchanged exploit revert, and revoking only the routine operator's `DealVaultFactory` role prevents creation of the rogue child.

An earlier proof also reached the same loss by letting the rogue child owner self-grant strategic authority and mint unbacked inventory, but that additional authority is unnecessary.

### Affected code

- `packages/smart-contracts/script/deploy.mainnet.s.sol`: routine operator-role grants
- `packages/smart-contracts/src/MasterVault/TenkaFundingFactory.sol`: arbitrary child ownership and automatic registration
- `packages/smart-contracts/src/DealVault/DealVaultFactory.sol`: arbitrary child ownership and automatic registration
- `packages/smart-contracts/src/DealVault/DealVault.sol`: owner transfer authority over third-party holders
- `packages/smart-contracts/src/MasterVault/MasterVault.sol`: trusts every registered FundingVault
- `packages/smart-contracts/src/MasterVault/FundingVault.sol`: trusts every registered DealVault and posts the par bids
- `packages/smart-contracts/src/MasterVault/MMFManager.sol`: helper configuration and request processing
- `packages/smart-contracts/src/MatchingEngine.sol`: routine execution of queued FundingVault orders

### Impact

Medium. A malicious authorized routine operator can cause direct, permanent loss from either the MasterVault or FundingVault. The demonstrated DealVault branch can consume the FundingVault's full idle USDC balance in one ordinary settlement window without false Oracle data, adverse pricing, virtual issuance, an approved receiver, or attacker capital. The required operator authority lowers likelihood despite the potentially large loss.

## Recommendation

Remove routine factory authority wherever vault creation is intended to require independent approval. Keep both FundingVault and DealVault creation under the intended timelocked multisig or admin Safe, revoke any existing routine grants, and add deployment tests that fail when a routine operator can create attacker-owned trusted vaults.

Do not treat factory registration alone as investment approval. Require an independent allowlist before a MasterVault or FundingVault can allocate to or trade against a newly registered child.

Remove a DealVault owner's ability to transfer shares from arbitrary third-party holders. If a recovery transfer is required, restrict it to separately authorized governance with explicit holder consent and an auditable reason.

## Developer response

Mitigated through deployment authority separation in [PR #320](#), commit [85beebc9](#).

FundingVault creation is assigned to the multisig and DealVault creation to the one-day timelock, rather than the routine operator. Temporary deployer roles are revoked and canonical FundingVault ownership is transferred to the multisig.

Commit [333e6665](#) adds handoff assertions. This mitigation does not introduce an investment allowlist or remove DealVault owner recovery-transfer authority.

### 2.6.6 Revoked KYC holders can liquidate DealVault positions through the MatchingEngine

The MatchingEngine validates the KYC status of each order's `receiver`, but it does not validate the DealVault seller identified by the ask order's `payer`. A holder whose KYC token has been frozen or revoked can therefore continue controlling and liquidating the position by directing the sale proceeds to another KYC address.

## Technical Details

Both posting and secondary-market matching check only the receiver:

```
1 // MatchingEngine.postOrder()
2 _isKYC(_signedOrder.order.receiver);

4 // MatchingEngine.matchOrders()
5 _isKYC(_bidMatch.signedOrder.order.receiver);
6 _isKYC(_askMatch.signedOrder.order.receiver);
```

For a DealVault ask, however, settlement transfers shares directly from `ask.payer`, not from `ask.receiver`:

```
1 DealVault(_bidOrder.instrument).transferDeal(
2     _askOrder.payer,
3     _bidOrder.receiver,
4     uint96(_bidOrder.trancheId),
5     _instrumentAmount - instrumentFee
6 );
```

`DealVault.transferDeal()` authorizes the `MatchingEngine` as the caller but does not validate the current KYC status of `_from`. Consequently, the following path succeeds through the normal `PRE_WINDOW` flow:

1. A KYC-approved seller acquires `DealVault` shares.
2. The seller's KYC token is subsequently frozen or revoked, for example after the wallet is reported compromised.
3. The old wallet posts or signs an ask with itself as `payer` and another KYC address as `receiver`.
4. A normal KYC and BLP-whitelisted buyer posts a crossing bid.
5. The orders are matched because both receivers pass KYC validation.
6. The `DealVault` shares leave the revoked wallet and the stablecoin proceeds are sent to the seller-selected KYC receiver.

## Impact

Medium. KYC revocation is intended to prevent the revoked holder from continuing to control or liquidate its `DealVault` position. Because the seller's KYC status is not checked, the revoked wallet can nevertheless dispose of its entire position through the ordinary secondary-market path, potentially affecting its full value without malicious operator behavior. This defeats both compromised-wallet recovery and compliance-driven asset restrictions, while allowing the proceeds to be redirected to another KYC address. Exploitation does, however, require access to a KYC proceeds receiver and sufficient matching liquidity.

## Recommendation

In the paired secondary-market branch of `MatchingEngine.matchOrders()`, validate the ask payer after determining whether the instrument is a `MasterVault`:

```
1 if (!isMasterVault) {  
2     _isKYC(_askMatch.signedOrder.order.payer);  
3 }
```

Performing this check during matching ensures that a `DealVault` ask cannot execute if the seller's KYC is revoked after the order was posted, while preserving the existing payer/receiver separation for purchases and `MasterVault` orders.

## Developer Response

Mitigated in [PR #320](#), commit [7eb34298](#). `DealVault` secondary settlement now checks the ask payer's current KYC, preventing revoked or frozen sellers from routing proceeds to another eligible receiver. Tests cover signed orders, revocation after posting, and valid payer/receiver separation. `MasterVault` behaviour remains unchanged.

## 2.7 Low Findings

### 2.7.1 ERC-1155 callback gas griefing can block Bookbuild processing

A KYC-approved contract bidder can consume nearly all gas during its ERC-1155 receipt callback while still returning the correct acceptance value. This causes the outer Bookbuild processing transaction to run out of gas and revert, leaving the same hostile bid at the front of the queue on every retry.

#### Technical Details

`DealVaultBookbuild.processBids()` processes the highest-priority bid and calls `DealVault.stakeFor()` inside `try/catch`. Minting the bidder's DealVault shares invokes `onERC1155Received()` with nearly all remaining gas.

If the receiver consumes that gas and then returns successfully, `stakeFor()` succeeds and the existing failure-handling branch does not run. The Bookbuild subsequently runs out of gas before it can finish the batch and persist its progress. The entire transaction rolls back, restoring the hostile bid as the mandatory queue head. Retrying or processing a smaller batch cannot skip it.

#### Impact

The attacker can block allocation for every later bidder in the affected funded round. The attacker earns nothing and must commit capital through a KYC-approved address. Administrators can restore processing by revoking the bidder's KYC and retrying, so the issue is classified as Low.

#### Recommendation

Forward a fixed, tested amount of gas to `stakeFor()`. If the call exceeds that limit, use the existing failure path to remove the bid, transfer its recovered assets to Recovery, and continue processing the round.

#### Developer Response

Acknowledged and accepted. Extremely unlikely and can burn KYC nft if ever needed

### 2.7.2 Zero-NAV turbo claims revert on yield-only distributions

`DealVault.claim(0)` always calculates how many turbo shares to burn after processing a claim. When both the tranche's clean NAV and the claimable principal are zero, this calculation divides by zero and prevents a FundingVault from collecting an otherwise valid yield-only distribution.

#### Technical Details

After `_claim()` determines and transfers the user's yield and principal, tranche-zero claims invoke `_processTurboSharesBurn()`:

```
1 (uint256 yield, uint256 primitive) = _claim(msg.sender, _trancheMask);
2 if (_trancheMask == 0) _processTurboSharesBurn(msg.sender, primitive);
```

The burn calculation uses the sum of clean NAV and claimed principal as its denominator:

```
1 uint256 cleanNav = _getNonStaleCleanNav(0);
2 uint256 previousCleanNav = cleanNav + _primitive;
3 uint256 sharesToBurn = balanceOf(_user, 0) * _primitive / previousCleanNav;
```

The oracle accepts a zero tranche NAV, and `updateYield()` permits a positive yield distribution with zero principal. In that state, `cleanNav == 0` and `_primitive == 0`, causing a division-by-zero panic. The revert rolls back the preceding yield transfer and claim accounting.

The FundingVault cannot bypass this path: its claim function calls `DealVault.claim(0)`, while the operator-assisted `claimFor()` and `claimBatch()` functions reject tranche zero.

## Impact

Low. A FundingVault cannot collect funded yield from a zero-NAV turbo position until principal or NAV becomes nonzero, or the DealVault is finalized. The affected value may be meaningful, but the required zero-NAV plus yield-only state is expected to be uncommon.

## Recommendation

Return early from `_processTurboSharesBurn()` when `_primitive == 0`, after validating the turbo owner and pending raised amount. No shares need to be burned for a yield-only claim.

## Developer Response

Acknowledged and accepted. Our intended lifecycle requires an updated, nonzero turbo-tranche NAV before processing yield-only claims. We accept the remaining limitation that yield-only claims revert when both NAV and claimable principal are zero. No code change planned.

### 2.7.3 Partial Agora swaps leave vault funds exposed through the public swap function

`AgoraHelper.depositStep()` and `redeemStep()` transfer the full requested input from the calling vault to the helper before executing an exact-input Uniswap V3 swap. If the pool reaches the configured price limit without consuming all of that input, the helper accepts the swap based only on its output and leaves the unspent vault funds in the helper contract. Because `swap()` is publicly callable and sends its output to the caller, those residual funds can subsequently be converted for an arbitrary caller.

## Technical Details

The MMF manager executes the helper steps by `delegatecall`. Each step therefore begins by transferring the requested input from the manager to the immutable helper address:

```

1 IERC20(inputToken).safeTransfer(THIS, _amount);
2 (uint256 amount0, uint256 amount1) = AgoraHelper(THIS).swap(_amount, zeroForOne);

```

`swap()` uses an almost-unbounded price limit:

```

1 uint160 sqrtPriceLimitX96 = _zeroForOne ? MIN_SQRT_RATIO + 1 : MAX_SQRT_RATIO - 1;
2 (amount0, amount1) = IUniswapV3PoolActions(UNISWAP_V3_POOL)
3   .swap(msg.sender, _zeroForOne, uint256(_amount), sqrtPriceLimitX96, "");

```

An exact-input Uniswap V3 swap may stop at its price limit after consuming less than `_amount`. The positive token delta returned by the pool identifies how much input was actually consumed, but neither helper step validates that delta. `_checkSwap()` instead checks only whether the nominal output is within two basis points of the requested input:

```

1 if (_receivedAmount < _amount) {
2   uint256 maxSlippage = _amount * MAX_BPS_SLIPPAGE / 10000;
3   if (_receivedAmount < _amount - maxSlippage) revert ErrSlippageExceeded();
4 }

```

A partial fill can therefore pass whenever its output satisfies this threshold. The difference between `_amount` and the positive input delta remains at `AgoraHelper`, outside the originating vault's holdings and request accounting.

The residual is exposed because `swap()` has no caller restriction, names `msg.sender` as the pool's recipient, and pays the pool from the helper during the callback:

```

1 function swap(uint256 _amount, bool _zeroForOne) external returns (uint256 amount0,
2   uint256 amount1) {
3   // ...
4   (amount0, amount1) = IUniswapV3PoolActions(UNISWAP_V3_POOL)
5     .swap(msg.sender, _zeroForOne, uint256(_amount), sqrtPriceLimitX96, "");
6 }
7
8 function uniswapV3SwapCallback(uint256 amount0Delta, uint256 amount1Delta, bytes calldata)
9   external {
10    if (msg.sender != UNISWAP_V3_POOL) revert ErrCallerNotUniswapV3Pool();
11    if (amount0Delta > 0) IERC20(TOKEN0).safeTransfer(UNISWAP_V3_POOL, uint256(
12      amount0Delta));
13    if (amount1Delta > 0) IERC20(TOKEN1).safeTransfer(UNISWAP_V3_POOL, uint256(
14      amount1Delta));
15 }

```

After the pool price moves away from the boundary, whether through ordinary rebalancing or deliberate manipulation, an arbitrary account can call `swap()` in the residual token's input direction. The pool then pulls that input from `AgoraHelper`, while sending the resulting output directly to the caller.

## Impact

Low.

## Recommendation

In both helper steps, verify that the positive input-side delta returned by the pool equals `_amount`, reverting the entire operation on a partial fill. Alternatively, measure the helper's input-token balance before and after the swap and return any unconsumed amount to the originating vault.

Additionally, restrict `swap()` to authorized MMF managers so any tokens accidentally held by the helper cannot be spent for an arbitrary recipient.

## Developer Response

Mitigated in [PR #320](#), commit [79dacfeb](#). After a swap, the helper returns its remaining input-token balance to the caller within the same transaction, covering both deposit and redemption directions. Public `swap()` access remains unchanged as an accepted design choice.

### 2.7.4 Reservation replacement can authorize two allocations

`reserveFor()` overwrites a user's existing reservation without checking whether it was consumed while the replacement transaction was pending. A user can therefore front-run a reservation update, consume the old allocation, and then consume the replacement after the operator's transaction executes.

## Technical Details

`takeReservation()` reads and clears the current reservation before funding the corresponding DealVault stake:

```
1 uint256 amount = reservedFor[_vault][_tranche][msg.sender];
2 reservedFor[_vault][_tranche][msg.sender] = 0;
3 // transfer funds and stake
```

Meanwhile, `reserveFor()` unconditionally writes the new value:

```
1 reservedFor[_vault][_tranche][_user] = _amount;
```

If an operator broadcasts a transaction replacing an existing reservation, the reserved user can consume the old value first. The pending update then writes the new value into the cleared slot, allowing the user to fund both allocations.

## Impact

Low. The user can exceed the allocation intended by the strategic operator, but must fully fund both stakes. The DealVault remains backed, so the consequence is limited to bypassing allocation policy rather than causing insolvency or direct loss.

## Recommendation

Require reservation updates to include the expected current value and revert if it has changed. Alternatively, clear the old reservation in a confirmed transaction before creating the replacement.

## Developer Response

Acknowledged and accepted. Partners would have access to reservations. Very unlikely that a partner is malicious for more shares of a DealVault.

### 2.7.5 MasterVault shares can be transferred to and liquidated by non-KYC addresses

`MasterVault` inherits unrestricted ERC-20 transfer functions, while its transfer hook enforces only the blacklist. Consequently, a KYC-approved holder can transfer shares directly to an address that has never received a KYC token, bypassing the `MatchingEngine` receiver check. The non-KYC holder can subsequently redeem the shares by directing the proceeds to another KYC-approved address.

## Technical Details

`BaseVault._update()` is the only restriction applied to share transfers:

```
1 function _update(address _from, address _to, uint256 _value) internal virtual override {
2     _checkBlacklisted(_from);
3     _checkBlacklisted(_to);
4     super._update(_from, _to, _value);
5 }
```

No KYC check is performed. Direct `transfer`, `transferFrom`, and ERC-3009 authorized transfers can therefore place MasterVault shares with any non-blacklisted address without going through the `MatchingEngine`.

During primary redemption, `_checkOneWayOrder()` validates only the signed order's `receiver`:

```
1 _isKYC(_matchingOrder.signedOrder.order.receiver);
```

It does not validate the ask `payer`. `_reduceTokenBalance()` can therefore pull shares from a non-KYC payer into the `MatchingEngine`; the transfer succeeds because `BaseVault` only checks the blacklist, and `MasterVault.redeem()` then burns those shares and sends the redemption assets to the KYC-approved receiver.

An end-to-end sequence is:

1. Alice acquires MasterVault shares while KYC-approved.
2. Alice calls `MasterVault.transfer(bob, shares)`, where Bob has no KYC token. The transfer succeeds.

3. Bob approves the `MatchingEngine` and signs a primary redemption ask with Bob as `payer` and a KYC-approved address as `receiver`.
4. An operator settles the order normally. Bob's shares are pulled and redeemed, and the nominated KYC receiver receives the assets.

The proof of concept in `test/audit/MasterVaultNonKycTransfer.t.sol` demonstrates a non-KYC address receiving and liquidating 100,000 MasterVault shares through this path.

## Impact

Low. A KYC holder can transfer the full economic exposure represented by its MasterVault shares to a non-KYC party without privileged cooperation or use of the controlled secondary market. The non-KYC holder can then liquidate the position through an ordinary operator-executed redemption by selecting a KYC proceeds receiver. This defeats KYC eligibility and revocation controls and permits OTC ownership changes outside the `MatchingEngine`'s fee and market controls. However, no direct third-party theft occurs, and on-protocol liquidation requires access to a KYC-approved receiver.

## Recommendation

Enforce current KYC status for ordinary MasterVault share senders and recipients in the transfer hook, while explicitly exempting minting, burning, and any protocol custody contracts that must hold shares. Alternatively, disable direct ERC-20 transfers and require all ownership changes to use the controlled `MatchingEngine` flow.

Independently recheck the MasterVault ask payer's KYC status at execution time for primary redemptions and secondary sales. Execution-time validation is necessary because KYC can be revoked after an order is posted or its shares are escrowed.

## Developer Response

Acknowledged and accepted. Intent to allow non-kyc users to hold mastervault shares is to allow potential defi integrations in the future.

### 2.7.6 Sparse tick scanning can require 101 refund transactions

`DealVaultBookbuild.processRefunds()` scans every price tick between the highest submitted bid and tick zero, including empty ticks. A bidder can submit at the maximum valid tick and force the operator to send 101 refund transactions before a valid tick-zero bidder is refunded and the round closes.

## Technical Details

Submitting a bid records its tick as the round's `highestBidTick`:

```
1 if (_tick * BIPS_INCREMENT > BIPS_MAX) revert ErrInvalidTick();
2 // ...
3 if (_tick > bookbuild.highestBidTick) bookbuild.highestBidTick = _tick;
```

`processRefunds()` performs at most `L00PS` iterations per call. An empty queue still consumes an iteration before the cursor moves down by one tick:

```
1 uint256 iterations = 0;
2 uint256 tick = bookbuild.highestBidTick;
3 while (iterations++ < L00PS) {
4     (Node memory node, uint256 index) = bookbuild.bidQueues[tick].peek();
5     address user = node.user;
6     if (user == address(0)) {
7         if (tick == 0) {
8             bookbuild.state = bbState.CLOSED;
9             return;
10        }
11        tick--;
12        continue;
13    }
14    // refund populated node
15 }
16 bookbuild.highestBidTick = tick;
```

The maximum valid tick is 5,000, while one call examines only 50 positions. A single high-tick bid therefore makes the refund cursor traverse thousands of empty queues before reaching ordinary bids at tick zero.

With one bid at tick 5,000 and a victim bid at tick zero, the round remains in `REFUNDING` after 100 successful `processRefunds()` transactions, the cursor has only reached tick one, and the victim has received nothing. Transaction 101 finally refunds the victim and closes the round. The high-tick bidder recovers its full bid. With both bids at tick zero, both users are refunded and the round closes in one transaction.

This is distinct from the previously discussed minimum-bid issue. Both bids satisfy the configured minimum amount; the delay results from scanning the sparse price-tick range.

### Affected Code

- `packages/smart-contracts/src/DealVault/DealVaultBookbuild.sol`: `_commit()`, `processRefunds()`
- `packages/smart-contracts/src/DealVault/Queue.sol`: queue traversal

### Impact

Low. An attacker can delay refunds and round closure while imposing substantial operational transaction costs. User funds remain in custody and can eventually be recovered by continuing to call `processRefunds()`.

### Recommendation

Track populated ticks in an ordered structure or store a pointer to the next populated tick. Refund processing should scale with the number of live bids rather than the numeric distance between their ticks.

## Developer Response

Acknowledged and accepted. With a \$10,000 post-premium minimum, a fully allocated maximum-tick bid generates a \$50,000 premium; full cancellation instead incurs a \$1,500 penalty plus forfeited MMF yield. We accept the additional processing costs and delays, including failed raises where bids are refunded without either charge.

### 2.7.7 Compromised operator can drain FundingVault assets through max-spread round trips

A compromised FundingVault operator can repeatedly buy DealVault shares from a cooperating KYC holder above net asset value (NAV) and sell them back below NAV. Each round trip transfers value from the FundingVault to the cooperating account even though every Oracle value is correct.

## Technical Details

FundingVault initializes `maxOrderSpread` to 1,000 basis points, or 10%:

```
1 function initialize(address _owner, address _registry, address _token) external  
  initializer {  
2   __MMFManager_init(_registry, _owner, _token);  
3   maxOrderSpread = 1000;  
4 }
```

The same operator can post both bid and ask orders for the FundingVault. `postOrder()` only verifies that each individual price remains within the configured spread:

```
1 if (_signedOrder.order.orderType == OrderType.BID) {  
2   if (_signedOrder.order.price > maxOrderSpread + BIPS_BASE) revert  
   ErrOrderPriceExceeded();  
3   // ...  
4 } else {  
5   if (_signedOrder.order.price < BIPS_BASE - maxOrderSpread) revert  
   ErrOrderPriceExceeded();  
6   // ...  
7 }
```

The check permits the FundingVault to buy at 110% of NAV and later sell the same position at 90% of NAV. The MatchingEngine operator also chooses which compatible orders to match. There is no independent authorization of the FundingVault's economic terms and no cumulative loss limit across related trades.

During `PRE_WINDOW`, the FundingVault must be BLP-whitelisted because it is the bid payer. During `SETTLING`, the BLP check does not apply. Moving the engine into `SETTLING` requires the strategic role, which the team plans to place behind a multisig, but that call only opens the settlement window. Once it is open, the normal MatchingEngine operator can execute individual matches without another on-chain multisig approval.

A compromised operator and cooperating KYC holder can therefore perform the following cycle:

1. The FundingVault buys the holder's DealVault shares at the upper spread boundary.

2. The FundingVault sells those shares back at the lower spread boundary.
3. The parties repeat the cycle while the FundingVault has liquidity.

At the default 10% spread, each completed cycle transfers value out of the FundingVault.

## Recommendation

Enforce price-direction limits for FundingVault orders. A FundingVault bid should not buy DealVault shares above NAV, and a FundingVault ask should not sell them below NAV. This still permits the FundingVault to buy positions at a discount and sell them at NAV or a premium while preventing the demonstrated loss-making round trip.

If the protocol must permit FundingVault purchases above NAV or sales below NAV, enforce an independently authorized cumulative loss budget rather than relying only on the maximum spread for each individual order.

## Developer Response

The team confirmed that the path is possible and that operator compromise leading to asset theft is in scope. The team plans to place the strategic role that opens `SETTLING` behind a multisig and noted that a FundingVault would need BLP status to buy during `PRE_WINDOW`. Independent monitoring may also trigger emergency mode when unsafe orders are observed.

The remaining question is whether the multisig approves each exact FundingVault match or only opens the ordinary settlement window, after which the normal operator can execute matches without further on-chain approval.

### 2.7.8 Negative MMF returns can lock bookbuild bids and subscribed rounds

`DealVaultBookbuild` records each bid's original asset amount separately from the ERC-4626 shares received from the configured `MmfVault`. Several settlement paths assume those shares can never become worth less than the original bid. If the staging vault realizes a loss, bidders may be unable to cancel while the raise is open, and a sufficiently subscribed round can become stuck in `PROCESSING` without an available refund transition.

## Technical Details

During cancellation, `_processCancelCommit()` calculates the MMF yield as:

```
1 uint256 mmfPenalty = (currentAssetValue - originalBidAmount) * bidAmount /  
    originalBidAmount;
```

If `previewRedeem(node.amount) < originalBidAmount`, the subtraction underflows and `cancelCommit()` reverts.

The state transition has the same underlying accounting mismatch. `advanceState()` selects `PROCESSING` by comparing the nominal `totalBidAmount` against `targetRaiseAmount`; it does not consider the current redeemable value of the staging-vault shares. `processBids()` then redeems the shares but continues to use the recorded gross bid:

```
1 uint256 assets = IERC4626(stagingVault).redeem(node.amount, address(this), address(this));
2 _stakeFor(_vault, _tranche, node.user, postPremiumAmount);
3 feesCollected[asset] += bidAmount - postPremiumAmount;
4 IERC20(asset).safeTransfer(node.user, assets - bidAmount);
```

For a zero-premium winning bid, staking the original amount fails because the bookbuild received fewer assets. For a premium bid whose loss can still be covered by the premium, `assets - bidAmount` instead underflows. The transaction reverts and restores the dequeued node and redeemed shares. Because cancellations require `RAISING` or `RERAISING` and refunds require `REFUNDING`, there is no built-in path out of `PROCESSING` after this failure.

This condition can arise through the intended MMF flow. `AgoraHelper` accepts up to two basis points of negative swap slippage while `getAssetFromShares()` values the received MMF token at 1:1. Consequently, swapping the staging asset into and back out of the MMF token can reduce `MmfVault.totalAssets()` while its ERC-4626 share supply remains unchanged.

A proof of concept modeled a decrease in `MmfVault` assets after a fully subscribed bid. The bidder's cancellation reverted with arithmetic panic `0x11`; after `advanceState()`, `processBids()` reverted due to insufficient assets; the round remained in `PROCESSING`, the queue was unchanged, and `processRefunds()` reverted with `ErrNotRefunding`.

## Impact

Low.

## Recommendation

Handle negative MMF returns explicitly. Use a saturating calculation for positive yield instead of subtracting the original bid unconditionally. Before or during processing, compare redeemable assets with the obligations represented by the bid and define how losses are allocated among the stake, premium, and refund.

If the realized assets cannot support the required allocation, provide a controlled transition from `PROCESSING` to a loss-adjusted refund state. Consider tracking the ERC-4626 shares associated with each round so `advanceState()` can evaluate current redeemable value rather than nominal bid accounting alone.

## Developer Response

Acknowledged, won't fix.

## 2.8 Gas Savings Findings

None.

## 2.9 Informational Findings

### 2.9.1 Harden deployment and vault-creation validation

The deployment and vault-creation tooling omits several configuration checks and finalization steps. These can be addressed before transactions are submitted.

#### Bookbuild rounding buffer

The Bookbuild deployment does not fund the raise-token buffer expected by the ERC-4626 rounding remediation.

Transfer the intended buffer to `DealVaultBookbuild` during deployment and verify its balance afterward.

#### Duplicate DealVault currencies

`DealVaultFactory.createVault()` accepts duplicate currency addresses. A duplicate can cause the same currency balance to be counted more than once.

Add a uniqueness check to the script or backend that prepares each vault-creation transaction. For example:

```
1 error DuplicateToken(address token);

3 function _validateUniqueTokens(address[] memory tokens_) internal pure {
4     for (uint256 i; i < tokens_.length; ++i) {
5         for (uint256 j; j < i; ++j) {
6             if (tokens_[i] == tokens_[j]) {
7                 revert DuplicateToken(tokens_[i]);
8             }
9         }
10    }
11 }
```

Run `_validateUniqueTokens(tokens)` before submitting the `createVault()` transaction. When executed as local script or backend validation, this check is not part of the on-chain transaction and adds no gas cost. It provides an inexpensive safeguard against accidental misconfiguration.

#### FundingVault authority handoff

The deployment transfers ownership of the other principal contracts but leaves the canonical `FundingVault` owned by the deployer. The deployer also retains its temporary `OPERATOR_ROLE`.

After configuration is complete:

1. Transfer the canonical `FundingVault` to the intended timelock or multisig.
2. Revoke the deployer's temporary `OPERATOR_ROLE`.
3. Verify the final owner, the intended operator assignment, and the absence of deployer authority.

## Missing commit operator

The deployment does not grant `COMMIT_OPERATOR_ROLE`, which is required to use `commitFor()`.

Grant the role to the intended account during deployment and verify the assignment. If sponsored commits are intentionally disabled, document that decision and omit the role deliberately.

## Recommendation

Implement the deployment, validation, and post-deployment checks described above.

## Developer Response

Mitigated in [PR #320](#), commit [333e6665](#). Each raise/re-raise now transfers 100,000 raw units of the raise token from the strategic operator into Bookbuild, funding the buffer when a round opens rather than at deployment.

The factory rejects duplicate currencies, and the mainnet script asserts final ownership, removal of deployer roles, and retention of the intended FundingVault operator. Regression tests cover buffer transfers and duplicate tokens. `COMMIT_OPERATOR_ROLE` remains deliberately unassigned because sponsored commits are not currently enabled.

### 2.9.2 Remove the unused `NAV_PENDING` state

`EngineState` includes `NAV_PENDING`, but no code path uses it. The team confirmed that the state is unused and should be removed.

## Recommendation

Remove `NAV_PENDING` and its unused references.

## Developer Response

Addressed in [PR #320](#), commit [30c9e308](#). The unused `NAV_PENDING` enum member has been removed. `EngineState` now contains `NULL`, `PRE_WINDOW` and `SETTLING`; consequently, `SETTLING` has ordinal 2.

### 2.9.3 Outstanding reservations can capture previously accrued yield

A user with an outstanding bookbuild reservation can exercise it immediately before a pending yield distribution and receive part of the yield that accrued before their shares existed.

## Technical Details

`DealVaultBookbuild.takeReservation()` has no expiry, issuance epoch, or yield record-date restriction. It calls `DealVault.stakeFor()`, which accounts the recipient at the current interest index and mints their shares. A subsequent `DealVault.updateYield()` distributes the pending yield across the now-enlarged total supply:

```
1 td.updateAccounting(_user, balanceOf(_user, _trancheId));
2 _mint(_user, _trancheId, shares, "");

4 // Later, in updateYield()
5 td.distributeInterests(_yield);
```

Consequently, a reservation holder can observe an `updateYield()` transaction, front-run it with `takeReservation()`, and immediately claim their pro-rata share of that distribution. A proof of concept demonstrates that when the incumbent and newly issued positions are equal, the new investor captures half of the pending yield.

## Impact

Informational. Although the behaviour is exploitable, both its likelihood and practical impact are very low because it requires a pre-existing reservation and favourable transaction ordering, while the attacker's principal remains invested.

## Recommendation

Monitor reservation executions around yield distributions, particularly cases where a user enters immediately before `updateYield()` and claims shortly afterwards.

## Developer Response

Acknowledged and accepted. The exploit is valid but unlikely. Reservation will be for partners and its usage will be agreed with involved parties.

### 2.9.4 EIP-7702 delegation lets a malicious operator exceed the Bookbuild target

A malicious Bookbuild operator can use EIP-7702 to delegate its EOA to contract code that implements an ERC-1155 receiver callback. Because the delegated code executes in the operator account's context, it retains the same address and `OPERATOR_ROLE`. When Bookbuild shares are minted to the operator, the callback can reenter `processBids()` and cause the round to allocate more than its approved target.

## Technical Details

`processBids()` copies the remaining target into memory and does not write the reduced value back to storage until after `DealVault.stakeFor()` returns. However, `stakeFor()` mints ERC-1155 shares to the winning bidder, which invokes the receiver callback before the reduced target is saved.

If the winning bidder is the operator's delegated EOA, the receiver callback can call `processBids()` again. The reentrant call passes `onlyOperator`, reads the original target from storage, and allocates another bid. When the outer call resumes, it completes using its stale in-memory target. Both bids can therefore be allocated even though only one fits within the approved raise.

### Impact

The malicious operator can force a bidder who should have received a refund into an illiquid DealVault position and make the vault raise more capital than approved. The behavior requires deliberate abuse by the trusted Bookbuild operator and does not demonstrate theft or insolvency, so it is reported as Informational.

### Recommendation

Make `processBids()` non-reentrant.

### Developer Response

Acknowledged and accepted. No code change planned.

## 2.9.5 DealVault Does Not Enforce Matching Token Decimals

### Technical Details

The `DealVault` supports multiple ERC20 tokens through the `tokens` array and explicitly documents that all supported tokens must use the same number of decimals:

```
1 /// @dev Every token in `_tokens` must use the same number of decimals
```

However, `initialize()` does not validate this invariant before storing the provided tokens:

```
1 if (_tokens.length == 0) revert ErrInvalidToken();
2 tokens = _tokens;
```

This is important because the vault aggregates the raw token amounts without performing any decimal normalization:

```
1 function _getRaisedAmount(uint96 _trancheMask) internal view returns (uint256
   raisedAmount) {
2     uint256 tokenCount_ = tokens.length;
3     for (uint256 i = 0; i < tokenCount_; i++) {
4         raisedAmount += trancheAmountRaised[_trancheMask][tokens[i]];
5     }
6 }
```

### Impact

Informational.

## Recommendation

Add to the vault or the factory the following check:

```
1 uint8 canonicalDecimals = IERC20Metadata(_tokens[0]).decimals();  
  
3 for (uint256 i = 1; i < _tokens.length; ++i) {  
4     if (IERC20Metadata(_tokens[i]).decimals() != canonicalDecimals) {  
5         revert ErrInvalidTokenDecimals();  
6     }  
7 }
```

## Developer Response

Mitigated in [PR #320](#), commit [60d85ad9](#). `DealVaultFactory.createVault()` now rejects tokens whose decimals differ from the canonical asset. Regression tests cover matching and mismatched decimals.



## Tenka Smart Contract Security Audit

Completed 2026-09-11